

## DETERMINATION OF THE EQUIVALENCE OF ALGEBRAIC EXPRESSIONS

### Technical Field of the Invention

The present invention relates generally to transformation of algebraic expressions into a standard form, and in particular, to the equivalence of original and transformed expressions. The present invention relates to a method and apparatus for performing such transformation, and also to a computer program product including a computer readable medium having recorded thereon a computer program for performing such transformation.

### Background Art

Modern compiler initiated code optimisation techniques attempt to detect common sub-expressions and evaluate them only once, whenever such opportunities are possible to exploit. Most compilers manage to do this only in a limited sense, and this ability varies from compiler to compiler. For example, it is sometimes recommended that common sub-expressions be placed at the beginning of the expression and/or enclosed in parentheses, and that they be written in the same way each time they occur. This recommendation is based upon the fact that a compiler will generally not recognise the common sub-expression in such simple expression pairs as:

$$T = U + X + Y \text{ and } V = (X + Y) + Z$$

or

$$T = U + X + Y \text{ and } V = Z + X + Y$$

or

$$T = U + X + Y \text{ and } V = X + Y + Z$$

Many compilers do better if the same statements are organised in one of the following ways:

$$T = X + Y + U \text{ and } V = X + Y + Z$$

or

$$T = U + (X + Y) \text{ and } V = Z + (X + Y)$$

or

$$T = U + (X + Y) \text{ and } V = X + Y + Z$$

Naturally, if all computer programmers wrote their expressions in a universally recognised format and were consistent in their application of that format, then the task of the compiler would be simplified. However, human nature being what it is, this is an impossible aim. Furthermore, over time a given computer program will require modifications, corrections and the like to cope with changed external requirements, detected errors, and so on. Such changes will not always be written by the same programmer, and a later programmer may have only a rudimentary idea of the way the original (or former) programmer has expressed algebraically identical functions or rules. Thus, when the amended program comes to be recompiled, it is particularly advantageous if the compiler can recognise recently written expressions as being equivalent to earlier written expressions.

### **Disclosure of the Invention**

It is an object of the present invention to substantially overcome, or at least ameliorate, one or more disadvantages of existing arrangements.

According to a first aspect of the present invention there is disclosed a method of determining, in a computer environment, the equivalence, if any, of two algebraic expressions for use in compiler optimisation of source code and like computing tasks, said method comprising the steps of:

- (a) recasting said expressions into a form of one or more token pairs arranged sequentially in a string, each said token pair comprising an operator followed by an operand;
- (b) reducing said strings in accordance with a set of predetermined simplifying rules; and
- (c) comparing the reduced strings by matching, to detect equivalence of the two algebraic expressions.

According to another aspect of the invention, there is provided an apparatus adapted to determine, in a computer environment, the equivalence, if any, of two algebraic expressions for use in compiler optimisation of source code and like computing tasks, said apparatus comprising:

- (a) recasting means for recasting said expressions into a form of one or more token pairs arranged sequentially in a string, each said token pair comprising an operator followed by an operand;
- (b) reduction means for reducing said strings in accordance with a set of predetermined simplifying rules; and
- (c) comparison means for comparing the reduced strings by matching, to detect equivalence of the two algebraic expressions.

According to another aspect of the invention there is provided a computer program product including a computer readable medium having recorded thereon a computer program for determining, in a computer environment, the equivalence, if

any, of two algebraic expressions for use in compiler optimisation of source code and like computing tasks, said program comprising:

- (a) recasting process steps for recasting said expressions into a form of one or more token pairs arranged sequentially in a string, each said token pair comprising an operator followed by an operand;
- (b) reduction process steps for reducing said strings in accordance with a set of predetermined simplifying rules; and
- (c) comparison process steps for comparing the reduced strings by matching, to detect equivalence of the two algebraic expressions.

The present invention relates to determining if two syntactically correct algebraic expressions are equivalent or not. Among its applications are compiler initiated optimisation of source code (where it is desirable to recognise multiple occurrences of invariant common sub-expressions in code blocks so that such sub-expressions can be evaluated once, and the result used in all the instances of the sub-expression in the block); simplification of algebraic expressions; etc.

### **Brief Description of the Drawings**

A preferred embodiment of the present invention will now be described with reference to the drawings, in which:

Fig. 1 shows a process flow diagram for a preferred embodiment of the present invention;

Fig. 2 depicts evolution of an algebraic expression during the process of Fig. 1;

Fig. 3 shows the preconditioning sub-process in Fig. 1 in more detail; and

09738095 120100

Fig. 4 is a schematic block diagram of a general purpose computer upon which the preferred embodiment of the present invention can be practiced.

### **Detailed Description including Best Mode**

Where reference is made in any one or more of the accompanying drawings to steps and/or features, which have the same reference numerals, those steps and/or features have for the purposes of this description the same function(s) or operation(s), unless the contrary intention appears.

The principles of the preferred method described herein have general applicability to compilers, text editors, database management software, and the like. However, for ease of explanation, the steps of the preferred method are described with reference to algebraic expressions, and in particular, equivalence of such expressions. However, it is not intended that the present invention be limited to the described method.

Fig. 1 depicts an expression simplification process 116 illustrating a preferred embodiment of the present invention in which an algebraic expression is simplified through the mechanism of identifying, and eliminating, redundant equivalent sub-expressions. In the first instance, it is assumed that the algebraic expression input at 122 for testing and processing in regard to equivalence is syntactically (ie grammatically) correct.

A convention adopted for the present embodiment is that all variable names in an expression are composed of lower case characters only. In this context, the underscore is a reserved character used only by the embodiment to modify a variable's name. The use of the underscore is described in more detail with reference to Fig. 3, in particular in relation to handling variables raised to a negative power.

The syntactically correct expression is subjected to a preconditioning sub-process 102 which will be described in more detail in relation to Fig. 2. Thereafter, the preconditioned expression is resolved into token pairs in a token pair resolution sub-process 104. After the pre-conditioning sub-process 102, the resulting expression (or string) is conveniently termed a *simple expression* or *simple string* 402 (see Fig. 2 for a summary of terminology). This may be viewed as a succession of token pairs 404 --<operator><operand> --. A token pair 404 whose <operator> is a \* is called a *product token pair* 406, and a token pair whose <operator> is + or - is called a *sum token pair* 408. A sum token pair and all its contiguously succeeding product token pairs are regarded as constituting a *subgroup* 410. Essentially the subgroups represent the terms in the simple algebraic expression, and the simple expression is a concatenation of the subgroups. The expression simplification process 116 is then directed to an exponent decoding and restoration sub-process 106 where exponents, having been coded in the pre-conditioning sub-process 102, are decoded and restored. Thereafter, operand tokens within each sub-group are rearranged in an operand rearrangement sub-process 108. The process 116 is then directed to a simplification sub-process 110. String matching among sub-groups is then performed in a string matching sub-process 112, after which the matching sub-groups are combined in a combination sub-process 114. Thereafter, the expression simplification process 116 is directed to a decision sub-process 118 which determines whether any matches remain to be performed. If they don't, then the process 116 is directed in accordance with the "NO" arrow to the end sub-process 120. If matches do still remain to be processed, then the process 116 is directed in accordance with the "YES" arrow back to the string matching sub-process 112, and so on.

Fig. 2 shows a partial evolution of the algebraic expression being simplified in the process 116 of Fig. 1, in particular, reviewing aspects of the terminology which is used. The output 124 of the preconditioning sub-process 102 is seen to comprise a simple expression or a simple string as depicted by a dashed inset 402. This simple expression 402, after passing through the token resolution sub-process 104, takes the form of token pairs 404. These pairs are either product token pairs 406, or alternatively sum token pairs 408. As noted above, a sum token pair 408 and all its contiguously succeeding product token pairs 406 are regarded as constituting a subgroup 410.

Turning to Fig. 3, the preconditioning sub-process 102 is shown in more detail. Commencing with the incoming process flow depicted by an arrow 122, in a blank deletion sub-process 200, blanks (or spaces) in the expression, if found, are deleted. Thereafter, all brackets in the expression are removed in a bracket removal sub-process 202 by carrying out the necessary operations to enable the expression to be written without brackets, for example:

$$a*(x-b) \text{ becomes } a*x-a*b$$

and

$$((a+b)) \text{ becomes } a+b$$

etc.

Thereafter, in the event that an expression does not commence with a unitary operator, the unitary operator + (plus) is inserted at the start of the expression in a unitary operator sub-process 204. Thus, for example:

$$a+b*c-d/f \text{ becomes } +a+b*c-d/f$$

Variables raised to a positive integer power are written out as multiplications of the variable in the following power reformulation sub-process 206. Thus, for example:

$x^n$  becomes  $x * x * \dots * x$  [The variable  $x$  appearing  $n$  times]

Likewise, a variable  $x$  raised to a negative integer power is written out as multiplications of the variable  $x\_$ . Thus, for example:

$x^{-n}$  becomes  $x\_ * x\_ * \dots * x\_$  [ $x\_$  appearing  $n$  times]

Note that  $x\_$  is used as the variable name instead of  $x$ . For its significance, refer to the description for the division sub-process 212 below.

Fractional powers are handled by an appropriate function, such as the `pow()` in C and C++. For more detail in regard to the handling of functions, see the description relating to **Handling of Aliases and Functions** below.

At this point, the expression is then able to be written as a string in the following form:

$\langle \text{unitary operator} \rangle \langle \text{operand} \rangle \langle \text{operator} \rangle \langle \text{operand} \rangle \dots \langle \text{operator} \rangle \langle \text{operand} \rangle$

where the unitary operator is either + (plus) or - (minus), and each operator is one of + (plus), - (minus), \* (multiplication) or / (division).

It is noted that  $\langle \text{unitary operator} \rangle$  is one of the unitary operators, which are operators which act on a single operand. The + and - operators are two such operators. Thus +a and -b are valid expressions, where a and b are operands.  $\langle \text{operator} \rangle$  is one of the binary operators, which are operators which act on two operands. The +, -, \*, / are such operators. Thus a+b, a\*b are valid expressions. Note that the + and -



We need consider only unsigned constants since they are treated on par with the variables. The constants are represented by an exponential notation in an e-format in the following constant reformulation sub-process 208 as follows:

where: [unsigned number] is a  $n$ -digit number comprising only digits and  $n$  is a prefixed integer greater than 0

[unsigned exponent] is a  $m$ -digit number comprising only digits and  $m$  is a prefixed integer greater than 0.

25 =  $0.25 \times 10^2$  becomes .250000e>02

$0.025 = 0.25 \cdot 10^{-1}$  becomes `.250000e<01`

where we have assumed  $n = 6$  and  $m = 2$ . Note that any constant will be represented by a string of constant length  $m+n+3$  characters in the e-format. Here  $e[e\text{-sign}][\text{unsigned exponent}]$  represents the quantity 10 raised to the power  $[e\text{-sign}][\text{unsigned exponent}]$ ,

which must be multiplied to the number represented by `[unsigned number]` to get the actual constant.

All `+` (plus) operators are then substituted, in a sign-reformulation sub-process 210 with the string `+1*` so that `+` becomes `+1*`. Similarly, all `-` (minus) operators are substituted with the string `-1*` so that `-` becomes `-1*`. (Actually, the `1`s are encoded in their e-format.) Thus, for example:

`a+b` becomes `a+.100000e>01*b`

and

`a+b-c*d` becomes `a+.100000e>01*b-.100000e>01*c*d`

Division by a constant (eg `c`) is then replaced in a division reformulation sub-process 212 by multiplication by the reciprocal ( $C=1/c$ ) whilst division by a variable (eg `x`) is replaced by multiplication by the reciprocal of the variable (eg `*x_ = 1/x` where `x_` is the variable `x` suffixed by an underscore). Thus, for example:

`y/c` becomes `y*C` and `a/x` becomes `a*x_`

Once this preconditioning process 102 has been carried out, as noted previously, the resulting expression (or string) is conveniently termed a simple expression or simple string 402.

Returning to Fig. 1, the simple expression 402 can now be viewed, or resolved, in the token pair resolution sub-process 104 as a succession of token pairs 404 `--<operator><operand>` -- beginning with the first operator token. The first operator token in the simple expression is guaranteed to be either `+` or `-`. A token pair whose `<operator>` is a `*` is called a *product token pair* 406, and a token pair whose `<operator>` is `+` or `-` is called a *sum token pair* 408. A sum token pair and all its contiguously succeeding product token pairs are regarded as constituting a *subgroup*

Thereafter, in the exponent decoding/restoration sub-process 106, in all the operand tokens, which are constants, the exponent sign  $>$  or  $<$  is decoded and restored, respectively, to  $+$  or  $-$ . The operand tokens, including the tokens which hold constant operands, in each subgroup 410 are now rearranged in ascending order in the operand token rearrangement sub-process 108, with each operand token treated as a character string.

When strings comprising individual characters are to be ordered, the ordering can be performed in a straightforward manner having regard, individually, to the ASCII representation of each character. For example, given the list of variables “a, a,

c, d, b" (having corresponding ASCII representations "97, 97, 99, 100, 98"), the ordered list takes the form "a, a, b, c, d" in accordance with an ascending order being imposed on the corresponding ASCII representations. When character strings to be ordered comprise multiple characters, ordering can be performed in a similar manner. Accordingly, (i) represent each variable eg "ab" as a concatenated ASCII representation "97|98" (the character | being used as a delimiter for the sake of this description only, ie, it is not used in the method itself) (ii) order the variables so that 1<sup>st</sup> characters in the strings are in ascending order in accordance with their corresponding ASCII representations (iii) considering variables having a common first character, order the variables so that the 2<sup>nd</sup> characters in the strings are in ascending order, (iv) considering variables having common first two characters, order the variables so that the 3<sup>rd</sup> characters in the strings are in ascending order, ..., and so on. Thus, for example, if the variables a, ab, a\_, aa, ba (having corresponding ASCII representations 97, 97|98, 97|95, 97|97, 98|97) are sorted in ascending order, they will appear in the order a, a\_, aa, ab, ba based on the ASCII value of the individual characters, ie 97, 97|95, 97|98, 98|97.

It is noted that "a" can be considered as "97" or "97|0". It is also noted that no other variable will be able to place itself between "a" and "a\_", if a and a\_ are present in the list of variables to be sorted.

In a further example, if the list of variables to be sorted are a, b, c, a, a, ba, a\_, c, a\_, then when this list is sorted in ascending order it will appear as a, a, a, a\_, a\_, b, ba, c, c.

When numbers are treated as character strings, such strings are also sorted according to the ASCII value of the characters appearing in the string. For example, the numbers .500000e+00 and .120000e+02, when sorted in ascending order will

appear in the order .120000e+02, .500000e+00, even though as numbers they are in exactly the opposite order. This is because the character 1 has a lower ASCII number than the character 5. The reason that all the numbers bunch at the beginning of an expression when the operands are sorted is because all variable names are required to begin with an alphabet (ie an "alpha" as against a "numeric" or other character), and the ASCII value of all the alphabets is greater than those of the characters 0 to 9 and the character representing the decimal point (whose ASCII number is 46).

The naming of variables using only lowercase alphabets does not restrict the use of the process, because no matter what convention is used for naming a variable, the expression can be put through a preconditioning step so that the variables therein are mapped to a corresponding new set of variables which are constructed using only lower case alphabets. The method can then be executed on this preconditioned expression, and in the final result the variables are mapped back to their original names. Alternatively, one may retain the variable's name but replace the underscore by some other reserved character whose ASCII number is smaller than any of the permitted characters in a variable name provided its introduction does not create any semantic ambiguities.

Since all the operators, except the leading operator, are all \* operators, which is a commutative operator, the rearrangement does not change the semantics of the subgroup. This is one of the reasons why the /x operator-operand pair was replaced with a contrived \*x\_\_ operator-operand pair.

In the simplification sub-process 110, advantage is taken of the rearrangement, by which the constant operand tokens will all bunch up at the beginning of the subgroup where they can be easily identified and replaced by a single constant. Also, in the rearrangement, a variable x and its counterpart reciprocal x\_\_, if it is present,

will appear as adjacent operands (ie x followed by x\_) a situation which is easily identifiable. This pair (along with their corresponding \* operators) can then be eliminated. (The underscore was deliberately used to bring about this ordering.) Thus, for example:

$$+.100000e+01*a*w*b*a_*.500000e+00$$

after arranging the operand tokens in ascending order becomes

$$+.100000e+01*.500000e+00*a*a_*b*w$$

Note the bunching of the constants and of the variables a and a\_. After consolidating the constants and eliminating the variables a and a\_ the previous expression becomes

$$+.500000e+00*b*w$$

Considering a further example, given the list of variables are a, b, c, a, a, ba, a\_, c, a\_, then when this list is sorted in ascending order it will appear as a, a, a, a\_, a\_, b, ba, c, c. Assuming for example that these variables represent the expression "+.100000e+00\*a\*a\*a\*a\_\*a\_\*b\*ba\*c\*c", the simplification subprocess 110 handles it thus: it sees "a\*a\_" in the expression and therefore deletes it. The expression now becomes "+.100000e+00\*a\*a\*a\_\*b\*ba\*c\*c". It looks at the expression again and sees yet another "a\*a\_" and therefore deletes it. The expression now becomes "+.100000e+00\*a\*b\*ba\*c\*c". No further simplification is possible in this example.

Continuing with the description in regard to Fig. 1, once the above operations are done, a string match is carried out in the string matching sub-process 112 among the subgroups. Thereafter, in the matching sub-group sub-process 114, all those subgroups, whose non-constant parts (that is, the third token onwards) match, are

combined by modifying the constant operand in one of the instances of the subgroup, and eliminating all other instances of the subgroup.

Note that all subgroups will have their leading operator-operand pair representing a signed constant and no other operand token in a subgroup will be a constant. This was ensured when all the + operators in the expression were replaced with +1\* and all the - operators in the expression were replaced with the string -1\*. If a constant operand in a subgroup is zero, that subgroup is eliminated.

After the above described recombination/elimination, the rematch and recombination/elimination processes are repeated by virtue of the decision sub-process 118 until no rematch is found. Then no further simplification of the expression is possible in its present representation. This is the reduced form of the original algebraic expression string. The nature of the operations is such that any two algebraic expressions, which are equivalent, will reduce to the same unique reduced string. If the reduced string is equivalent to a zero, then the string is set to "0.0".

A string match between the respective reduced strings of two given algebraic expressions then suffices to determine if the said two algebraic expressions are equivalent or not.

### **Handling of Aliases and Functions**

An aliased variable is a variable known by more than one name. It is analogous to a person being known by more than one name. In a computer program, all the different names of an aliased variable would refer to the same address in memory.

Accordingly, an aliased variable is easily handled by preparing a list of its aliases, alphabetically ordering the list and replacing all aliases with the name at the top of the ordered aliases list ( since all variable names are assumed to comprise lower

It is possible to extend the expression simplification process 116 to include functions by replacing the function by the string `funcname#`, where `funcname` is a suitably contrived name designating the function name, its parameter types, and its actual arguments. This turns the function into a *defacto* variable name. The character `#` (one can choose some other) is appended to avoid possible collisions between a `funcname` and a variable name. One particular care has to be taken: each of the actual function arguments must be converted to their respective reduced expression before creating the `funcname`. This will ensure that the same function when it appears with an equivalent parameter list at different places will reduce to the same `funcname`. Thus, for example, `sin(x)` and `sin(x + y/y - 1)` will return the same `funcname`. Thus, for example, `sin(x)` and `sin(x + y/y - 1)` will return the same `funcname` since their respective arguments `x` and `x+y/y-1` will both reduce to `+1.00000e+01*x`.

In the example, it is necessary to determine if:

are equivalent.

$$y + 4.0 * x - x - x - y - x * a / a + 2 * y / 2$$

The following sequence of operations is carried out:



1. remove all blanks [A blank is the same as a space.]
2. insert a leading + operator,
3. change all constants to the coded e-format,
4. replace the +'s with +1\* (ie +.100000e>01\*),
5. replace the -'s with -1\* (ie -.100000e>01\*),
6. replace /a by \*a\_, and
7. replace /2 with \*0.5 (ie \*.500000e>00).

This results in

+.100000e>01\*y+.400000e>01\*x-.100000e>01\*x-.100000e>01\*x-  
.100000e>01\*y- .100000e>01\*x\*a\*a\_+.200000e>01\*y\*.500000e>00

where the exponent + sign has been encoded to > (< is used for the - sign). Next, the subgroups are formed, which are listed below:

+.100000e>01\*y  
+.400000e>01\*x  
-.100000e>01\*x  
-.100000e>01\*x  
-.100000e>01\*y  
-.100000e>01\*x\*a\*a\_  
+.200000e>01\*y\*.500000e>00

In each of the subgroups we decode the exponent sign of the constants and sequence the operands in ascending order. The resulting subgroups are:

+.100000e+01\*y  
+.400000e+01\*x

-.100000e+01\*x

-.100000e+01\*x

-.100000e+01\*y

-.100000e+01\*a\*a\_\*x → .100000e+01\*x

+.200000e+01\*.500000e+00\*y → +.100000e+01\*y

In the rearranged subgroups we consolidate the constants (as done in the last subgroup here) and eliminate all occurrences of the pattern \*x\*x\_ (as done in the last but one subgroup where \*a\*a\_ is eliminated). The subgroups are now arranged in ascending order, ie alphabetically, (neglecting their constant part) as follows:

+.400000e+01\*x

-.100000e+01\*x

-.100000e+01\*x

-.100000e+01\*x

+.100000e+01\*y

-.100000e+01\*y

+.100000e+01\*y

Notice that the common subgroups have now bunched together, and a simple forward scan (from the third token onwards in the subgroups) allows them to be identified easily and consolidated leading to the final string.

+.100000e+01\*x+.100000e+01\*y

This is precisely the string we would have got for  $x + y$ .

The above described procedures enable algebraic expressions to be reduced to a unique form for easy comparison.

09728096 430400  
00F0E7 908E268

The foregoing describes only one embodiment of the present invention and modifications, obvious to those skilled in the art, can be made thereto without departing from the present invention.

The method of determining the equivalence of two algebraic expressions is preferably practiced using a conventional general-purpose computer system 300, such as that shown in Fig. 4 wherein the processes of Figs. 1 and 3 may be implemented as software, such as an application program executing within the computer system 300. In particular, the steps of method of determining the equivalence of two algebraic expressions are effected by instructions in the software that are carried out by the computer. The software may be divided into two separate parts; one part for carrying out the determination of algebraic equivalence, and another part to manage the user interface between the latter and the user. The software may be stored in a computer readable medium, including the storage devices described below, for example. The software is loaded into the computer from the computer readable medium, and then executed by the computer. A computer readable medium having such software or computer program recorded on it is a computer program product. The use of the computer program product in the computer preferably effects an advantageous apparatus for determining the equivalence of two algebraic expressions in accordance with the embodiments of the invention.

The computer system 300 comprises a computer module 301, input devices such as a keyboard 302 and mouse 303, output devices including a printer 315 and a display device 314. A Modulator-Demodulator (Modem) transceiver device 316 is used by the computer module 301 for communicating to and from a communications network 320, for example connectable via a telephone line 321 or other functional medium. The modem 316 can be used to obtain access to the Internet, and other

The computer module 301 typically includes at least one processor unit 305, a memory unit 306, for example formed from semiconductor random access memory (RAM) and read only memory (ROM), input/output (I/O) interfaces including a video interface 307, and an I/O interface 313 for the keyboard 302 and mouse 303 and optionally a joystick (not illustrated), and an interface 308 for the modem 316. A storage device 309 is provided and typically includes a hard disk drive 310 and a floppy disk drive 311. A magnetic tape drive (not illustrated) may also be used. A CD-ROM drive 312 is typically provided as a non-volatile source of data. The components 305 to 313 of the computer module 301, typically communicate via an interconnected bus 304 and in a manner which results in a conventional mode of operation of the computer system 300 known to those in the relevant art. Examples of computers on which the embodiments can be practised include IBM-PC's and compatibles, Sun Sparcstations or alike computer systems evolved therefrom.

Typically, the application program of the preferred embodiment is resident on the hard disk drive 310 and read and controlled in its execution by the processor 305. Intermediate storage of the program and any data fetched from the network 320 may be accomplished using the semiconductor memory 306, possibly in concert with the hard disk drive 310. In some instances, the application program may be supplied to the user encoded on a CD-ROM or floppy disk and read via the corresponding drive 312 or 311, or alternatively may be read by the user from the network 320 via the modem device 316. Still further, the software can also be loaded into the computer system 300 from other computer readable medium including magnetic tape, a ROM or integrated circuit, a magneto-optical disk, a radio or infra-

red transmission channel between the computer module 301 and another device, a computer readable card such as a PCMCIA card, and the Internet and Intranets including email transmissions and information recorded on websites and the like. The foregoing is merely exemplary of relevant computer readable mediums. Other computer readable mediums may be used without departing from the scope and spirit of the invention.

The method of determining the equivalence of two algebraic expressions may alternatively be implemented in dedicated hardware such as one or more integrated circuits performing the functions or sub functions of determining the equivalence of two algebraic expressions. Such dedicated hardware may include graphic processors, digital signal processors, or one or more microprocessors and associated memories.

#### **Industrial Applicability**

It is apparent from the above that the embodiment(s) of the invention are applicable to the computer and data processing industries.

The foregoing describes only one embodiment of the present invention, and modifications and/or changes can be made thereto without departing from the scope and spirit of the invention, the embodiment being illustrative and not restrictive.